USER FRIENDLY HIGH PRODUCTIVITY COMPUTATIONAL WORKFLOWS USING THE VISION/HPC PROTOTYPE

J.H. Unpingco
Ohio Supercomputer Center
Columbus, OH 43212

I. ABSTRACT

HPCs (high-performance computers) utilize multiple (e.g. hundreds or thousands of) processors to compute very large problems quickly by distributing the computation across many processors in parallel. This liberates problem conceptualization from the memory/storage constraints of a single desktop workstation. Unfortunately, the complexity of programming HPCs is off-putting for new users. Furthermore, most DoD users work from a Windows PC so that learning Unix well enough to parallel program is itself an obstacle. What is needed is a workflow by which simplifies the programming task in a familiar environment while leveraging the computational power of HPCs.

VISION (Sanner, 2002) is a freely available, Python-based, drag-and-drop visual programming environment that programming for drawing flowcharts that encapsulate the underlying programming complexity. This means that computations are strung together by dropping and connecting computational boxes on a canvas instead of writing source code files. This is important for productivity since productivity is dominated by the time spent programming versus the time spent analyzing results.

As a Python-based package, it is possible to embed parallel computing features from the open source iPython package into VISION to enable both visual programming and parallel execution on remote HPCs. This paper discusses the prototype we built at SSC-SD for a visual parallel programming workflow based on VISION and iPython for parallel computing using a Linux cluster as a backend and a Windows XP workstation as the front-end.

II. INTRODUCTION

Productivity in scientific computing is dominated by the time spent coding versus the time spent reflecting on computed results. Insight is driven by how quickly intuition can be vetted -- how fast a hunch can be tested while pursuing a particular line of thought. Parallel computing accelerates computing, but is notoriously

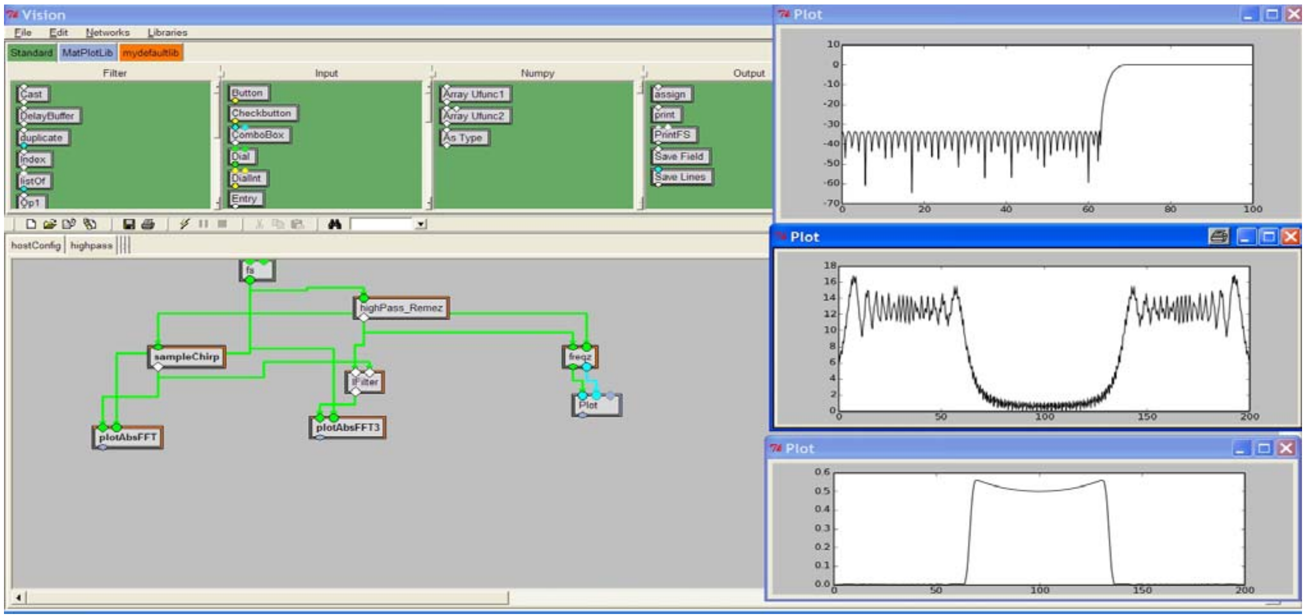


Fig. 1: VISION provides the drag-and-drop functionality for the network shown, which accomplishes a signal filtering task. The individual nodes represent code fragments that are connected to implement a particular computation. Everything is written in Python.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **DEC 2008**	2. REPORT TYPE **N/A**	3. DATES COVERED **-**
4. TITLE AND SUBTITLE **User Friendly High Productivity Computational Workflows Using The Vision/Hpc Prototype**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Ohio Supercomputer Center Columbus, OH 43212**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
See also ADM002187. Proceedings of the Army Science Conference (26th) Held in Orlando, Florida on 1-4 December 2008, The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**UU**	**4**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

complex and daunting for non-specialists.

Further, in a team research setting, a few individuals may have the requisite parallel programming skills and are relied upon by others for these tasks. This is a bottleneck and a deterrent to investigating new ideas. On the other hand, this prototype breaks tasks down into manageable computational units which others can easily manipulate while coincidently freeing up the team members who have parallel programming skills since they can focus on maintaining smaller computational units instead of the entire computation.

Thus, improving scientific productivity by facilitating parallel programming in a familiar Windows PC environment for both specialists and non-specialists was the main motivation for the VISION/HPC prototype. The basic idea is to run VISION on the user's Windows workstation and have the embedded parallel tasks execute on a remote backend. The key advantage is that the graphic intensive GUI runs on the local workstation (and is not served pixel-by-pixel through a slow network connection) and the intensive computation runs on the remote cluster (not overloading the local workstation). In the following, we describe in detail how this was accomplished and and examine a case-study that shows how the VISION/HPC workflow operates in practice.

III. Prototype Components

A. VISION

VISION is a freely available Python-based visual computing environment from the Scripps Research Institute in San Diego. It is part of a biomolecular visualization toolkit distributed by the Institute. It is entirely Python-based and all of the source code is available. VISION reduces programming tasks to dropping and connecting computing "nodes" onto a flowchart. Fig.1 shows a VISION network that accomplishes a signal filtering and visualization task.

Since VISION is entirely written in Python, it is compatible with all other Python modules. In particular, it works with the Scientific Python (SciPy) module, which includes a wide variety of scientific computing codes (e.g. matrix factorizations, data-fitting, statistical estimators). Furthermore, VISION comes with several libraries (i.e. sets of nodes) focused on specific categories (e.g. image processing).

There are two mechanisms for adding customized nodes to VISION. One can build nodes graphically by starting with the "generic" node available in the standard library. Alternatively, one can write Python scripts with input/output interfaces represented as interconnection "ports". Once these codes are developed, they become drag-and-drop usable in the flowchart.

Once a node is developed, VISION provides a text-editor interface for altering a node's internal code and other properties via the context menu (see Fig. 2). This makes it easy to quickly change the underlying computation and see immediate results. This is a very important feature for a visual programming environment and is unique to VISION; other environments require *everything* to be created visually, whereas VISION allows *both* visual programming and embedded text-based editing. This means that legacy codes do not have to be built visually, but can be pulled in through the text-editor

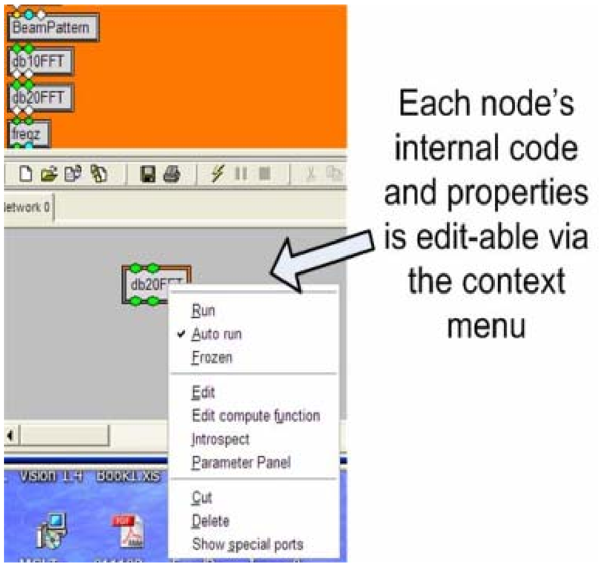
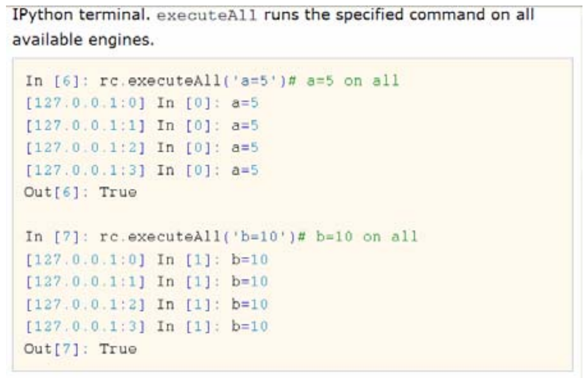


Fig. 2.: The embedded Python code in every node on the VISION canvas can be inspected an edited from the context menu.

and selectively exposed to the visual environment by choosing the input/output ports.

B. iPython

So far, we have discussed VISION as a tool for simplifying the construction of computing tasks, but not the parallel computing functions. That part is handled by



IPython terminal. executeAll runs the specified command on all available engines.

```
In [6]: rc.executeAll('a=5')# a=5 on all
[127.0.0.1:0] In [0]: a=5
[127.0.0.1:1] In [0]: a=5
[127.0.0.1:2] In [0]: a=5
[127.0.0.1:3] In [0]: a=5
Out[6]: True

In [7]: rc.executeAll('b=10')# b=10 on all
[127.0.0.1:0] In [1]: b=10
[127.0.0.1:1] In [1]: b=10
[127.0.0.1:2] In [1]: b=10
[127.0.0.1:3] In [1]: b=10
Out[7]: True
```

Fig. 3: Shows a sample iPython session where Python statements are interactively executed on remote processors.

iPython (Pérez, 2007) which is an open source Python-based development environment for *interactive* parallel computing as opposed to batch-based parallel computing. This means that, given an HPC configured for interactive use, users can move data between nodes, assign computing tasks, and monitor completion interactively. Fig. 3 shows an example of an iPython session where specific commands are executed all available nodes. This means no more waiting for results to be computed, stored, and then loaded since these are immediately in the same iPython workspace.

Further, iPython's parallel computing mechanisms can be used directly from a standard Python script, which thus allows it to be embedded into the VISION framework. Variables can be defined and shared between nodes using iPython. Thus, iPython has two modes of operation – interactive and embedded – and both are usable from within VISION, although we will only be using the embedded mode for this prototype[1].

C.Workflow Design

Given the mutual compatibility and open-design philosophy of Python embodied in VISION and iPython, the prototype was designed to call iPython commands from within VISION. It is important to understand which Python packages are going to be run on the backend (SSC-SD Linux cluster) as opposed to the local workstation (Windows PC) and to ensure that these are all appropriately configured beforehand. Thus configured, VISION can run on the PC and have its embedded nodes compute on the backend and then return the reduced data to the workstation for visualization or inspection. VISION comes with a one-click installer for Windows and is not necessary (although possible) to install VISION on the backend. IPython comes as a standard Python distributable and relies on the Twisted and Zope Python packages, which are available on both Windows PCs and Unix/Linux platforms.

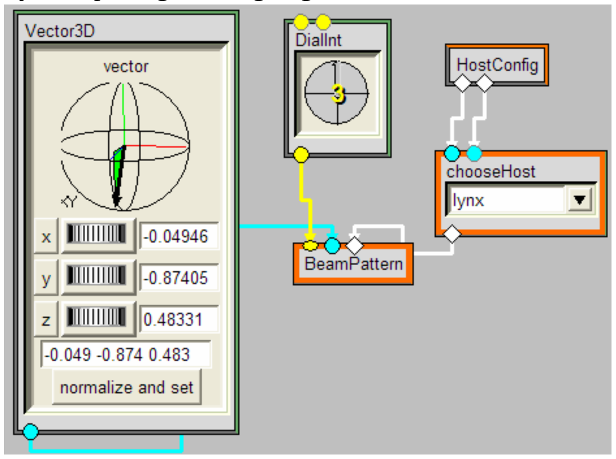


Fig. 5. The VISION network shown computes the beampattern for a given array configuration using the angle-of-arrival chosen by dragging the arrow on the unit sphere in the `Vector3D` widget. Multiple Linux clusters are available via the `chooseHost` node. The `DialInt` node tells the `BeamPattern` node how many pieces to divide the problem into so it can be distributed over the same number of processors.

```
from IPython.Shell import IPShellEmbed
ipshell = IPShellEmbed()
ip=ipshell.IP

x,y,z=xyzDir

rc=kernel.RemoteController((IP_Address, 10105))
rc.activate()
rc.getIDs()
ip.ipmagic('px import os')
ip.ipmagic('px os.chdir(os.environ["HOME"]+"/sshDemo")')

ip.ipmagic('px from scipy import arange')
ip.ipmagic('px import runDetect')
ip.ipmagic('px reload(runDetect)')
```

Fig.4. Python code showing how iPython is used within a script to setup the `runDetect` function, which computes the beampattern for the array.

D. Platforms

VISION and iPython were installed on the Windows XP workstation. SciPy and iPython and all subordinate dependencies were installed on the Linux cluster backend.

E. Case Study: Array Processing Beampattern

As a case study, we implemented an array processing beamforming problem. The problem is to compute the electromagnetic antenna pattern generated by sources at different arrival angles given certain array configurations. Depending on the level of detail and the size of the array, these patterns can be extremely computationally intensive. Fig. 4 shows the essential code used to setup the backend computations using iPython.

Note that the computationally intensive part derives from computing a set of physics equations and is accomplished in the `runDetect` function and that the code shown in Fig. 4 is just an interface to it. This is a good practice for modular design since the `runDetect` script can be developed and tested separately outside of the VISION/HPC framework. The main function of the code shown in Fig.4 is to setup the intermediate pieces from iPython (i.e., `ip.ipmagic`), which would be typed in during an interactive session. This is a key, but usually overlooked advantage to the two iPython modes because it allows the code shown above to be developed interactively and then pasted into a script.

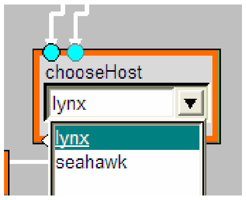


Fig. 6: Multiple Linux clusters can be used as backends for computation.

Fig. 5 shows the network for the beamforming problem. The `Vector3D` widget comes with the standard VISION toolset and provides the coordinates for

[1] Future develop plans include the ability to work in either mode, which would be useful for debugging.

a unit vector at a position on the sphere shown. The arrival angle of the source is selected on the unit sphere by dragging the arrow. The act of dragging the point shown on the widget provides a tactile connection to the problem and is important for intuition building. The resulting coordinates are input to the `BeamPattern` node and the `DialInt` node tells the `BeamPattern` node how many pieces into which the problem should be divided so that it can be distributed over the same number of processors

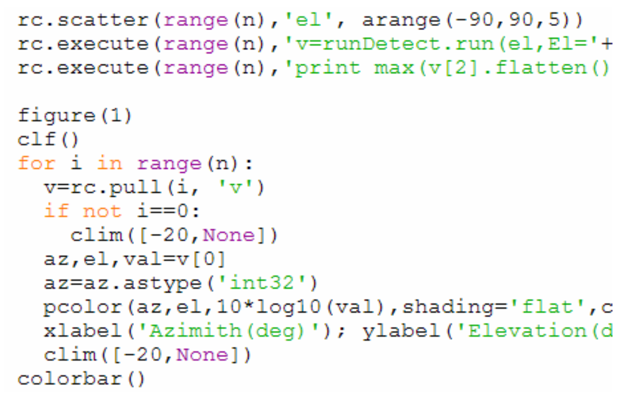
We used the `chooseHost` node (see Fig. 5) to switch the backend between two different SSC-SD clusters (Lynx, Seahawk) to demonstrate the flexibility of this approach. Once the host is chosen, the VISION/HPC flowchart executes the computation on that host.

Fig. 7 shows the remainder of the embedded code in the `BeamPattern` node, which includes the division of the computational task across a given number of processors. This is accomplished using the gather/scatter mechanism that is built into iPython. The next block merely pulls the various sub-pieces from the different processors and assembles the final graph shown in the next figure.

The graph in Fig. 8 shows the computed beampattern. The horizontal white lines delineate the computational stripes that have been computed on different processors based on the value of the `DialInt` node. This result is synthesized and presented on the Windows PC, so no high-bandwidth graphics were transmitted. In fact, only the commands to create the data shown in each of the three pieces was transmitted.

IV. Summary

We have established a Windows-to-HPC computational workflow to enhance productivity using the freely available VISION visual computing environment in connection with the parallel computing features of the iPython package. We examined the performance of this workflow using an array processing beamforming case study. The workflow prototype was designed so that the graphically intensive GUI ran on the local Windows PC workstation and the intensive computations ran on the remote Linux cluster. The reduced data product was returned to the local workstation and rendered there, thus saving the slow pixel-by-pixel transmission of the graphic through the computer network.



```
rc.scatter(range(n),'el', arange(-90,90,5))
rc.execute(range(n),'v=runDetect.run(el,El='+
rc.execute(range(n),'print max(v[2].flatten()

figure(1)
clf()
for i in range(n):
  v=rc.pull(i, 'v')
  if not i==0:
    clim([-20,None])
  az,el,val=v[0]
  az=az.astype('int32')
  pcolor(az,el,10*log10(val),shading='flat',c
  xlabel('Azimith(deg)'); ylabel('Elevation(d
  clim([-20,None])
colorbar()
```

Fig. 7: Remainder of code in the Beampattern node showing how the results are pulled in from different processors and assembled into the final graph.

V. Bibliography

Pérez, Fernando and Granger, Brian E., 2007: IPython: A System for Interactive Scientific Computing, *Comput. Sci. Eng.*, **9**,21-29.

Sanner M.F., Stoffler D. and Olson A.J. 2002. ViPEr, a Visual Programming Environment for Python. *Proceedings of the 10th International Python conference*. 103-115.

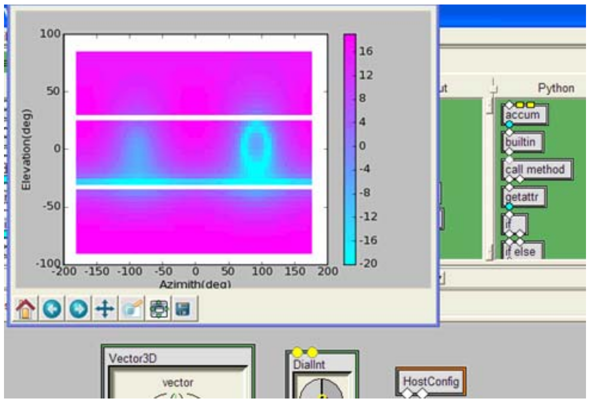


Fig. 8: The resulting beam pattern is shown. The three horizontal stripes indicate that three processors on the Linux cluster were used for the computation. The prototype distributes the computation across the three selected processors and reassembles the result on the local workstation. Note that the graphic is rendered on the local workstation and not transmitted pixel-by-pixel through the network connection (which would be slow).